

# Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

### Trabalho Prático 2

47283 : Ricardo Duarte Cardoso Bernardino (a47283@alunos.isel.pt)

47249: Miguel Henriques Couto de Almeida (a47249@alunos.isel.pt)

48270: Daniel Lopes Pina (a48270@alunos.isel.pt)

Relatório para a Unidade Curricular de Inteligência Artificial da Licenciatura em Engenharia Informática e de Computadores

Professor: Doutor Nuno Miguel da Costa de Sousa Leite

# **Contents**

Li	st of 1	Figures		V
Li	stage	ns		vii
1	Sud	oku So	lver	1
	1.1	Descr	ição do Problema	1
	1.2	Jogo .		1
		1.2.1	Descrição do Jogo	1
		1.2.2	Funcionamento do Jogo	2
	1.3	Algor	itmos	3
		1.3.1	Deapth-First-Search Iterative Deepening	3
			1.3.1.1 Algoritmo Base	4
			1.3.1.2 Função Objetivo	4
			1.3.1.3 Função Sucessora	5
		1.3.2	<i>A* Search</i>	6
			1.3.2.1 Algoritmo Base	7
			1.3.2.2 Função Objetivo	8
			1.3.2.3 Função Sucessora	8
			1.3.2.4 Função Heurística	9
		1.3.3	Simulated Annealing	10

iv *CONTENTS* 

Referêi	ncias			25
	1.4.2	Aprendi	zagem	22
	1.4.1	Resultac	los	22
1.4	Concl	usão		22
		1.3.4.4	Solução Final	18
		1.3.4.3	Mutação	17
		1.3.4.2	Reprodução	16
		1.3.4.1	Seleção	15
	1.3.4	Genetic A	Algorythm	15
		1.3.3.3	Algoritmo SA	12
		1.3.3.2	Função de Custo	11
		1.3.3.1	Cálculo da Temperatura	10

# **List of Figures**

1.1	Jogo em Curso	2
1.2	Boards que foram usados nos Testes	3
1.3	Resultados DFS-ID	6
1.4	Resultados A*	10
1.5	Resultados SA	14
1.6	Resultados GA	21
1.7	Resultados	22

# Listagens

1.1	Algoritmo Base - DFS ID	4
1.2	Função Objectivo - DFS ID	4
1.3	Função Sucessora - DFS ID	5
1.4	Algoritmo Base - A* Search	7
1.5	Função Objetivo - A* Search	8
1.6	Função Sucessora - A* Search	9
1.7	Função Heurística - A* Search	9
1.8	Cálculo da Temperatura - SA	10
1.9	Função de Custo - SA	11
1.10	Algoritmo SA	12
1.11	Algoritmo de Selecção - GA	15
1.12	Algoritmo de Reprodução - GA	16
1.13	Algoritmo de Mutação - GA	18
1.14	Algoritmo GA	19

1

# Sudoku Solver

## 1.1 Descrição do Problema

Neste 2º trabalho prático foi proposto a implementássemos algoritmos de pesquisa informados e não informados com o propósito de resolver um jogo de tabuleiro chamado *Sudoku*[3]. Além dos algoritmos de pesquisa, também era proposto que lidássemos e entendêssemos o conceito de espaço de estados dentro do âmbito dos algoritmos propostos.

# 1.2 Jogo

# 1.2.1 Descrição do Jogo

*Sudoku* é um jogo jogado por 1 jogador numa grid de 9x9. O objetivo final é formar um tabuleiro completo sem que haja repetição de números na mesma linha, coluna e subgrid.

1. Sudoku Solver 1.2. Jogo

### 1.2.2 Funcionamento do Jogo

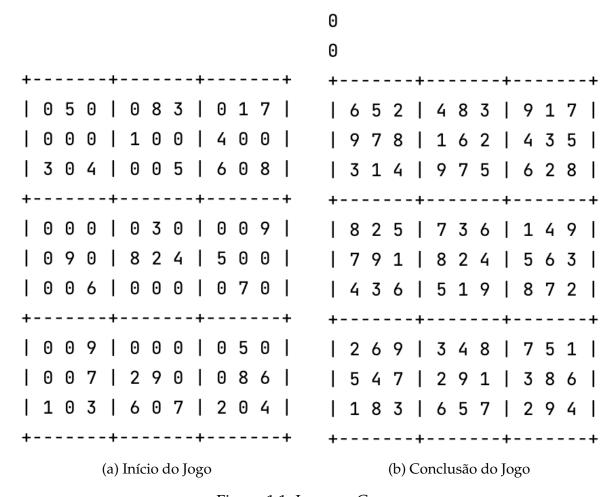
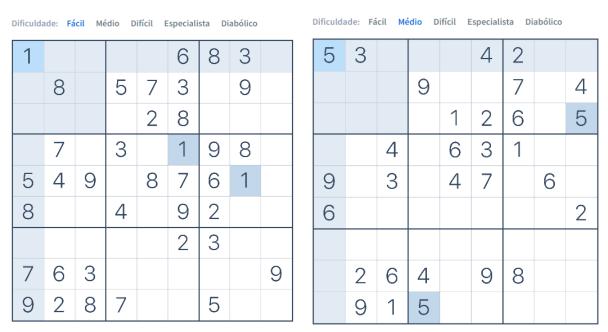


Figura 1.1: Jogo em Curso

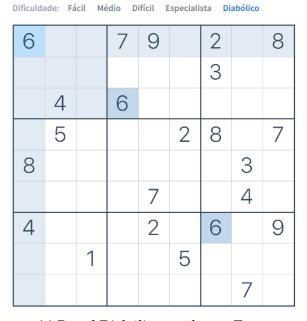
1. SUDOKU SOLVER 1.3. Algoritmos

# 1.3 Algoritmos



(a) Board Fácil usado nos Testes

(b) Board Médio usado nos Testes



(c) Board Diabólico usado nos Testes

Figura 1.2: Boards que foram usados nos Testes

## 1.3.1 Deapth-First-Search Iterative Deepening

O algoritmo *Depth-First Search (DFS)* é um algoritmo de pesquisa em grafo que percorre este explorando o máximo possível em profundidade. Isto significa que o algoritmo visita o primeiro vizinho de um vértice antes de visitar o segundo vizinho e

assim sucessivamente, aprofundando-se na pesquisa até atingir um vértice sem filhos, momento esse em que volta a trás (backtracking) para visitar os outros vértices. O Iterative Deepening é uma variação deste algoritmo, para maximizar a eficiência de procura, para grafos com grandes profundidades, incrementando a profundidade gradualmente, e realizando várias chamadas ao algoritmo base DFS por profundidade. Só após explorar totalmente uma certa profundidade é que o algoritmo avança para uma nova.

#### 1.3.1.1 Algoritmo Base

```
1 :- consult('path.pl').
2
  depth_first_iterative_deepening( Node, Solution) :-
    path( Node, GoalNode, Solution),
    goal(GoalNode).
5
6
7
   % ?- depth_first_iterative_deepening(a, Solution).
9
10
  path(Node, Node, [Node]). % Single node path
11
12
   path(FirstNode, LastNode | Path]):-
13
    path(FirstNode, OneButLast, Path), % Path up to one-but-last node
14
    s(OneButLast, LastNode),
                                   % Last step
15
    \+ member(LastNode, Path).
                                     % No cycle
16
17
18
  %? – path(a, Last, Path).
```

Listagem 1.1: Algoritmo Base - DFS ID

#### 1.3.1.2 Função Objetivo

A função objetivo averigua apenas se existem posições vazias no tabuleiro, sendo que, se existirem, ainda não foi concluída a resolução deste. Ao não existirem posições vazias no tabuleiro, é assim atingido o objetivo final do algoritmo, e concluído que todas as posições foram devidamente preenchidas.

```
1 goal(Board):-
```

1. SUDOKU SOLVER 1.3. Algoritmos

- empty\_positions(Board, List),
- 3 List == [].

Listagem 1.2: Função Objectivo - DFS ID

#### 1.3.1.3 Função Sucessora

A função sucessora é implementada através da simples lógica de selecionar a primeira posição vazia do tabuleiro, e gerar um número aleatório entre 1 e 9, visto que são as únicas possibilidades. Após a geração do número aleatório, é averiguada a presença de erros desta inserção, tentando validar a jogada realizada, falhando caso hajam.

- 1 s(Board, NextBoard):-
- 2 first\_empty\_position(Board, X/Y),
- 3 member(Num, [1,2,3,4,5,6,7,8,9]),
- 4 validate\_move(Board, X/Y, Num, NextBoard),
- 5 write("[DEBUG] -> "), writeln(NextBoard).

Listagem 1.3: Função Sucessora - DFS ID



Figura 1.3: Resultados DFS-ID

#### **1.3.2** *A\* Search*

A\* é um algoritmo de pesquisa em grafos ponderados, partindo de um nó inicial específico de um grafo, ele visa encontrar um caminho para o *goal* dado com o menor custo. O algoritmo mantém uma árvore de caminhos originados no nó inicial e estende esses caminhos aresta a aresta aprofundando a pesquisa até atingir um nó terminal (sem filhos), momento esse em que volta a trás (backtracking) para visitar os outros nós, combinando o custo acumulado até o momento (representado pela função "g(n)") com a heurística estimada (representada pela função "h(n)"), o A\* é capaz de selecionar nós terminais que têm o custo total mais baixo (representado por "f(n) = g(n) + h(n)"). Isso permite que o algoritmo explore os caminhos mais promissores primeiro, em vez de seguir uma busca cega em todas as direções.

#### 1.3.2.1 Algoritmo Base

```
bestfirst(Start, Solution) :-
    expand([], l(Start, 0/0), 9999, _, yes, Solution). % Assume 9999 is > any f-value
  expand(P, I(N, \_), \_, \_, yes, [N \mid P]):-
    goal(N).
6
7 expand(P, l(N, F/G), Bound, Tree1, Solved, Sol):-
    F = < Bound,
    (bagof(M/C, (s(N, M, C), \+ member(M, P)), Succ),
10
    succlist(G, Succ, Ts),
11
    bestf(Ts, F1),
12
    expand(P, t(N, F1/G, Ts), Bound, Tree1, Solved, Sol)
13
14
    Solved = never
15
16
    ).
17
  expand(P, t(N, F/G, [T | Ts]), Bound, Tree1, Solved, Sol) :-
18
    F = < Bound,
19
    bestf(Ts, BF), Bound1 = min(Bound, BF),
20
    expand([N | P], T, Bound1, T1, Solved1, Sol),
21
    continue(P, t(N, F/G, [T1 | Ts]), Bound, Tree1, Solved1, Solved, Sol).
22
23
24 expand( _, t(_, _, []), _, _, never, _) :- !.
  expand(_, Tree, Bound, Tree, no, _):-
    f(Tree, F), F > Bound.
26
27
  continue( _,_,_, yes, yes, Sol).
28
   continue(P, t(N, F/G, [T1 | Ts]), Bound, Tree1, no, Solved, Sol) :-
29
    insert(T1, Ts, NTs),
30
    bestf(NTs, F1),
31
    expand(P, t(N, F1/G, NTs), Bound, Tree1, Solved, Sol).
32
33 continue(P, t(N, F/G, [ | Ts]), Bound, Tree1, never, Solved, Sol) :-
    bestf(Ts, F1),
34
    expand(P, t(N, F1/G, Ts), Bound, Tree1, Solved, Sol).
35
36
37 succlist( _, [], []).
```

```
succlist(GO, [N/C | NCs], Ts):-
    G is GO + C,
39
    h( N, H),
40
    F is G + H,
41
    succlist(GO, NCs, Ts1),
    insert(l(N, F/G), Ts1, Ts).
43
44
   insert( T, Ts, [T \mid Ts]) :-
45
    f( T, F), bestf( Ts, F1),
46
    F = < F1, !.
47
   insert( T, [T1 | Ts], [T1 | Ts1]) :-
    insert(T, Ts, Ts1).
49
50
51 f( l(_, F/_), F).
52 f( t(_, F/_, _), F).
53
54 bestf([T | _], F):-
    f(T, F).
55
  bestf([], 9999).
```

Listagem 1.4: Algoritmo Base - A\* Search

#### 1.3.2.2 Função Objetivo

A função objetivo averigua apenas se existem posições vazias no tabuleiro, sendo que, se existirem, ainda não foi concluída a resolução deste. Ao não existirem posições vazias no tabuleiro, é assim atingido o objetivo final do algoritmo, e concluído que todas as posições foram devidamente preenchidas.

```
goal(Board):-
empty_positions(Board, List),
List == [].
```

Listagem 1.5: Função Objetivo - A\* Search

#### 1.3.2.3 Função Sucessora

A função sucessora é implementada através da simples lógica de selecionar a primeira posição vazia do tabuleiro, e gerar um número aleatório entre 1 e 9, visto que são as

únicas possibilidades. Após a geração do número aleatório, é averiguada a presença de erros desta inserção, tentando validar a jogada realizada, falhando caso hajam.

```
s(Board, NextBoard) :-
first_empty_position(Board, X/Y),
member(Num, [1,2,3,4,5,6,7,8,9]),
validate_move(Board, X/Y, Num, NextBoard),
write("[DEBUG] -> "), writeln(NextBoard).
```

Listagem 1.6: Função Sucessora - A\* Search

#### 1.3.2.4 Função Heurística

[2][1] A heurística é usada para orientar a busca do A\* e influencia a seleção dos nós. Ela fornece uma indicação de quão promissores são os nós não explorados em termos de proximidade do objetivo. Ao utilizar essa informação, o algoritmo pode priorizar a exploração de nós que têm uma heurística mais baixa, ou seja, que parecem estar mais próximos do objetivo. No caso do sudoku irá priorizar nós com o menor numero de espaços vazios possíveis.

```
h(Board, H):-
empty_positions(Board, List),
length(List, H).
```

Listagem 1.7: Função Heurística - A\* Search

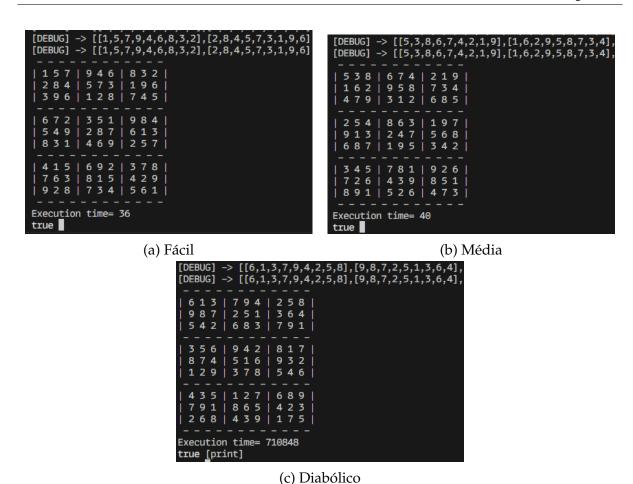


Figura 1.4: Resultados A\*

## 1.3.3 Simulated Annealing

#### 1.3.3.1 Cálculo da Temperatura

```
1 fun calculateStandardDeviation(numbers: List<Double>): Double {
2
     val mean = numbers.average()
     val variance = numbers.map { (it - mean) * (it - mean) }.average()
3
     return kotlin.math.sqrt(variance)
4
5 }
6
   fun calculateStartingTemperature(template: String): Double {
     val results = mutableListOf<Int>()
8
     val board = getBoardFromTemplate(template)
9
     repeat(200) {
10
       val filledBoard = board.fillWithRandValues()
11
12
       results.add(filledBoard.getTotalCost())
     }
13
```

```
return calculateStandardDeviation(results.map { it.toDouble() })

15 }
```

Listagem 1.8: Cálculo da Temperatura - SA

Esta função calcula a temperatura inicial para o algoritmo Simulated Annealing gerando aleatoriamente uma série de tabuleiros preenchidos com valores aleatórios com base no modelo fornecido. De seguida, calcula o custo total de cada tabuleiro gerado e armazena os resultados numa lista. Finalmente, retorna o desvio padrão dos resultados para ser usado como temperatura inicial na função de minimização.

#### 1.3.3.2 Função de Custo

```
fun getTotalCost(): Int {
    var rows = 0
    var cols = 0

for (index in 0 until BOARD_SIDE) {
    rows += getErrorsRow(index)
    cols += getErrorsColumn(index)
}

return rows + cols
}
```

Listagem 1.9: Função de Custo - SA

Esta função calcula o custo total de um tabuleiro que no contexto do algoritmo Simulated Annealing representa a medida que determina o quão bom é o tabuleiro atual em relação à solução desejada. Quanto maior o custo total, pior é o tabuleiro visto que estamos a tentar minimizar a temperatura.

A função getTotalCost() itera sobre todas as linhas e colunas do tabuleiro e calcula o número de erros em cada uma. Ela chama as funções getErrorsRow(index) e getErrors-Column(index) para obter o número de erros em cada linha e coluna, respetivamente. Essas funções retornam a contagem de erros para uma determinada linha ou coluna.

Ao percorrer todas as linhas e colunas, a função acumula os números de erros em rows e cols. Por fim, retorna a soma de rows e cols, que representa o custo total do tabuleiro.

O custo total é crucial no algoritmo Simulated Annealing, pois é usado para avaliar a qualidade das soluções durante o processo otimização.

#### 1.3.3.3 Algoritmo SA

```
fun SA(template: String): Board {
     val initialBoard = getBoardFromTemplate(template)
3
     initialBoard.print()
4
     var currentBoard = initialBoard.fillWithRandValues()
5
     var currentCost = currentBoard.getTotalCost()
7
     var temperature = calculateStartingTemperature(template)
8
     var stuckCounter = 0
10
     \mathbf{var} previousCost = 0
11
12
13
     while (currentCost > 0 && temperature > 0.0) {
        previousCost = currentCost
14
        for (iter in 1..ITERATIONS_PER_TEMPERATURE) {
15
          println(currentCost)
16
          val newBoard = currentBoard.generateNextBoard()
17
          val newCost = newBoard.getTotalCost()
18
          val diff = newCost - currentCost
19
20
          if (newCost < currentCost) {</pre>
21
            currentBoard = newBoard
22
            currentCost = newCost
23
          } else if (exp(-diff / temperature) > Random.nextDouble()) {
24
            currentBoard = newBoard
25
            currentCost = newCost
26
27
          }
28
29
        if (currentCost >= previousCost) {
30
          stuckCounter += 1
31
        } else {
32
          stuckCounter = 0
33
34
        if (stuckCounter > 100){
35
          //re-heat
36
          temperature += 2
37
```

```
38  }
39
40  temperature *= COOLING_RATE
41  }
42  return currentBoard
43 }
```

Listagem 1.10: Algoritmo SA

Esta função implementa o algoritmo Simulated Annealing. Recebe um Board no formato String como argumento e retorna um Board completo no final. O algoritmo começa por transformar a String num tabuleiro que possa ser usado. De seguida, o tabuleiro é preenchido com valores aleatórios e é calculado custo total do tabuleiro.

O algoritmo continua iterando enquanto o custo total não atingir zero e a temperatura for maior que zero. Durante cada iteração, um novo tabuleiro é gerado a partir do tabuleiro atual. O custo total do novo tabuleiro é calculado e comparado com o custo do tabuleiro atual e se o novo custo for menor, o tabuleiro atual é substituído pelo novo tabuleiro, caso contrário, o novo tabuleiro pode ser aceite com uma certa probabilidade, determinada pela função exponencial, pois é essa probabilidade que permite aceitar movimentos que aumentem o custo total, mas que também permitam escapar de mínimos locais.

O algoritmo também se o custo total permanece o mesmo por um certo número de iterações consecutivas (contador de "stuck"). Se isso ocorrer, o algoritmo aumenta a temperatura para "reiniciar"e permitir maior exploração.

A temperatura vai sendo reduzida de acordo com uma taxa de arrefecimento e o processo continua até que o custo total atinja zero ou que a temperatura caia para zero. Com temperatura a zero e a função de custo minimizada atingimos um board finalizado ou resolvido, que é o caso do nosso problema.



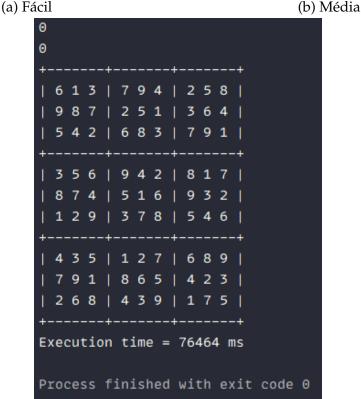


Figura 1.5: Resultados SA

(c) Diabólico

### 1.3.4 Genetic Algorythm

Os Algoritmos Genéticos são inspirados no princípio Darwiniano da evolução das espécies e na genética. São algoritmos probabilísticos que fornecem um mecanismo de busca paralela e adaptativa baseado no princípio de sobrevivência dos mais aptos e na reprodução. Neste sentido, a implementação do *Genetic-Algorithm* para efeitos de resolução de um jogo de Sudoku, é puramente baseado nos seus princípios fundamentais, que são adaptados às regras do jogo de sudoku, ou seja a seleção, seguida da reprodução, seguida da mutação.

#### 1.3.4.1 Seleção

A seleção dos pais pode ser realizada de diferentes formas, porém a abordagem decidida foi através do modo de torneio. O modo de torneio, ou tourney, seleciona X indivíduos, e compete-os uns contra os outros baseado-se nos seus scores, ou seja, do seu valor de fitness. No caso do jogo de sudoku, um maior score, indica um maior valor de fitness, pois o score indica o número de casas do tabuleiro que não esteja em conflito, ou seja, em erro, no tabuleiro em questão, sendo que um tabuleiro com menos erros é um individuo mais desejável. Neste torneio, são selecionados 5 indivíduos, e retornado aquele que tiver um maior score. Foi também implementada a seleção por roleta, selecionando indivíduos com probabilidades proporcionais ao seu score.

```
1
2 / * *
    * Seleção por Roleta. Seleciona um individuo com uma
       probabilidade proporcional ao seu score.
5 fun <T> fitnessProportionateSelection(scoredPopulation: Collection<Pair<Double, T
       >>): T {
     var value= scoredPopulation.sumOf { it.first } * random()
6
7
     for ((fitness, individual) in scoredPopulation) {
8
       value -= fitness
9
       if (value <= 0) return individual
10
11
12
13
     return scoredPopulation.last().second
14 }
15
16 / * *
```

```
* Modo de seleção por Torneio. Seleciona 5 individuos randomly
17
      da população, e retorna aquele com maior score
18
19 fun <T> sudokuSelectionTournament(scoredPopulation: Collection<Pair<Double, T
      >>): T {
    val tournamentSize = 5
20
21
    val tournament = scoredPopulation.shuffled().take(tournamentSize)
22
    val winner = tournament.maxByOrNull { it.first }!!.second
23
24
25
    return winner
26 }
```

Listagem 1.11: Algoritmo de Selecção - GA

#### 1.3.4.2 Reprodução

A Reprodução, para criação de filhos, através de dois pais, é realizada nesta implementação através da criação de novos tabuleiros (filhos), através da junção genética de dois tabuleiros diferentes (pais). A junção genética, ou crossover, é realizada pela função sudokuCrossover apresentada abaixo. Esta função gera um número aleatório entre 1 a 7, representando um índice de bloco do tabuleiro, para que haja uma divisão justa, não é gerado um índice em que nenhum bloco terá sido selecionado de um dos pais, ou seja, no mínimo (índice gerado é igual a 7, ou seja o 8º bloco) é sempre selecionado 1 bloco de um dos pais. De seguida, é retirado bloco a bloco, até atingir esse índice, o/os bloco/os do pai, e finalmente após esse índice, é retirado o/os bloco/os da mãe. A junção final será o filho (tabuleiro) gerado.

```
1 /**
2 * Faz o crossover entre 2 parentes, pora gerar um filho. 0
    filho resultante terá os primeiros n blocos do parent1,
3 * até um crosspoint gerado randomly, e os restantes blocos do parent2.
4 */
5 fun sudokuCrossover(parents: Pair<List<Cell>>, List<List<Cell>>>): List<List<Cell>>> (
6 val crossoverPoint = (0..7).random()
7 val child: MutableList<MutableList<Cell>> = mutableListOf()
```

```
9
     for (i in 0 until 9) {
        val row: MutableList<Cell> = mutableListOf()
10
        for (j in 0 until 9) {
11
          val cell = Cell(0) // Initialize every cell with the digit 0
12
          row.add(cell)
13
14
        child.add(row)
15
16
     println("Parent 1: ")
17
     Board(parents.first).print()
18
19
     println("Parent 2: ")
20
     Board(parents.second).print()
21
     var childBoard = Board(child)
22
23
     var blockPositions : List<Pair<Int,Position>>
24
     var block: List<Cell>
25
26
     val parent1 = Board(parents.first)
27
     val parent2 = Board(parents.second)
28
29
     for(i in 0 .. crossoverPoint){
30
       block = parent1.getBlock(i)
31
       blockPositions = parent1.getBlockPositions(i).toList()
32
       println(blockPositions.size)
33
       blockPositions.forEach {
34
         childBoard = childBoard.insertCell(it.second,block[i])
35
36
       }
    }
37
```

Listagem 1.12: Algoritmo de Reprodução - GA

#### 1.3.4.3 Mutação

A mutação dos filhos é realizada através da função *sudokuMutation* que seleciona aleatoriamente 1 bloco do tabuleiro desse filho. Neste bloco, como é sempre garantida a regra de não repetição de números dentro desta, é trocado aleatoriamente um número que não seja fixo, ou seja, que não tenha sido gerado pelo algoritmo, mas sim estava presente no estado inicial do tabuleiro dado ao algoritmo.

```
1 / * *
   * Seleciona randomly 1 bloco da board, e depois seleciona
      randomly 2 números não-fixos, que serão trocados um com o
      outro.
  fun sudokuMutation(individual: List<List<Cell>>) : List<List<Cell>> {
    val blockIndex = (0...8).random()
    var board = Board(individual)
    val blockPositions = board.getBlockPositions(blockIndex).toList()
7
     /**Tenta encontrar um valor desse bloco que não seja fixo**/
8
    var randomNum1 = (0 .. 8).random()
9
    var num1 = blockPositions[randomNum1]
10
    var value1 = board.get(num1.second)
11
    while(value1.fixed){
12
13
       randomNum1 = (0 .. 8).random()
       num1 = blockPositions[randomNum1]
14
       value1 = board.get(num1.second)
15
16
     /**Tenta encontrar um valor desse bloco que não seja fixo**/
    var randomNum2 = (0 ... 8).random()
18
    var num2 = blockPositions[randomNum2]
19
    var value2 = board.get(num2.second)
20
    while(value2.fixed | | value2 == value1){
21
       randomNum2 = (0 .. 8).random()
22
      num2 = blockPositions[randomNum2]
23
       value2 = board.get(num2.second)
24
    }
25
26
27
     /**Efetua a troca**/
    return board.insert(num1.second,num2.first).insert(num2.second,num1.first).board
28
29
```

Listagem 1.13: Algoritmo de Mutação - GA

#### 1.3.4.4 Solução Final

Agora com as 3 funções principais do *Genetic Algorithm* realizadas, com base num jogo de sudoku, a função principal do algoritmo é implementada com uma função *Run* que

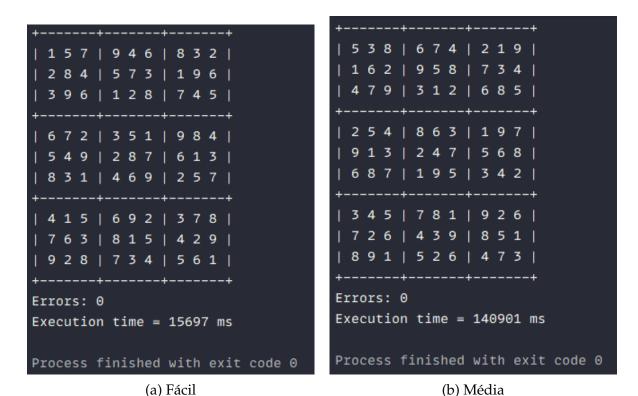
para 1000 epochs, ou seja 1000 ciclos, realiza o processo conjunto de seleção, reprodução e mutação, a cada ciclo, sob a população apresentada. Após estes ciclos de evolução genética, é esperado, sob uma certa probabilidade elevada, que o tabuleiro, ou filho, retornado por esta função, seja um que tenha um score total de 81, ou seja contenha 81 casas sem conflitos, um tabuleiro resolvido, sem erros.

```
1 class GA<T>(
     var population: Collection<T>,
     val score: (individual: T) -> Double,
3
     val cross: (parents: Pair<T, T>) -> T,
     val mutate: (individual: T) -> T,
     val select: (scoredPopulation: Collection<Pair<Double, T>>) -> T) {
6
7
     /**
8
         * @return O melhor individuo, após o número de epochs ter
       executado sob a população dada.
10
11
         * @param epochs - numero de epochs (ciclos de mutação)
12
         * @param mutationProbability - valor entre 0 e 1 que define
        a probabilidade do Filho ser mutado
13
     fun run(epochs: Int = 1000, mutationProbability: Double = 0.1): T {
14
       var scoredPopulation = population.map { Pair(score(it), it) }.sortedByDescending {
15
       it.first }
16
       for (i in 0..epochs)
17
         scoredPopulation = scoredPopulation
18
            .map { Pair(select(scoredPopulation), select(scoredPopulation)) }
19
            .map { cross(it) }
20
            .map { if (random() <= mutationProbability) mutate(it) else it }</pre>
21
            .map { Pair(score(it), it) }
22
            .sortedByDescending { it.first }
23
24
       return scoredPopulation.first().second
25
     }
26
27
28
29
  fun sudokuSolver(template : String) : List<List<Cell>> {
```

1. SUDOKU SOLVER 1.3. Algoritmos

```
val boardTemplate = getBoardFromTemplate(template)
31
     println("Initial Board: ")
32
     boardTemplate.print()
33
     println()
34
     println()
35
     val population = (1 .. 100).map {boardTemplate.fillWithRandValues().board}
36
37
     val algorithm = GA<List<Cell>>>(
38
       population,
39
       score = {81 - Board(it).getTotalCost().toDouble()},
40
       cross = ::sudokuCrossover,
41
       mutate = ::sudokuMutation,
42
       select = ::sudokuSelectionTournament
43
     )
44
     return algorithm.run()
45
46
47
  fun main(){
48
49
     //val result = binaryArrayExample()
50
     val result = sudokuSolver("
51
       .5..83.17...1..4..3.4..56.8....3...9.9.8245....6....7...9....5...729...86
       ")
52
     print("Best individual: ")
53
     result.forEach { print(it) }
54
55 }
```

Listagem 1.14: Algoritmo GA



+-----+
6 1 3	7 9 4	2 5 8
9 8 7	2 5 1	3 6 4
5 4 2	6 8 3	7 9 1
+-----+		
3 5 6	9 4 2	8 1 7
8 7 4	5 1 6	9 3 2
1 2 9	3 7 8	5 4 6
+-----+		
4 3 5	1 2 7	6 8 9
7 9 1	8 6 5	4 2 3
2 6 8	4 3 9	1 7 5
+-----+
Errors: 0

(c) Diabólico

Process finished with exit code 0

Execution time = 2825632 ms

Figura 1.6: Resultados GA

### 1.4 Conclusão

#### 1.4.1 Resultados

Para cada algoritmo foi testado o seu tempo de resolução consoante 3 dificuldades diferentes de puzzle (Fácil, Médio, e Diabólico). Os Resultados permitem comparar a eficiência de resolução de cada algoritmo perante o mesmo tipo de puzzle, permitindo concluir que para os puzzles mais fáceis (Fácil e Médio), o algoritmo  $A^*$  tem um menor tempo de resolução. Enquanto para os puzzles mais difíceis (Diabólico) o algoritmo *Simulated Annealing* tem um menor tempo de resolução.

G en eti c Algorithm - G A parameters					
Dificulty	Pop. Size	Iterations	Mutation Prob.	Errors	
Easy	500	2000	0,1	0	
Medium	2000	8000	0,1	0	
Diabolic	10000	10000	0,1	0	

(a) Parâmetros do Algoritmo Genético

Performance Analysis - Sudoku Solver					
Dificulty					
Language/Algorithm		Easy	Medium	Diabolical	
SWI-Prolog	DFS	2443	45961	2 720 313	
SWI-Prolog	A*	36	40	710848	
Kotlin	SA	1406	3677	76464	
Kotlin	GA	15697	140901	2825632	
	Note: Time expressed in millisecond				

(b) Análise de Performance

Figura 1.7: Resultados

### 1.4.2 Aprendizagem

Este trabalho prático apelou à nossa capacidade de implementar 4 algoritmos diferentes tanto em *Prolog* como em *Kotlin*. A observação da capacidade de resolução de cada

algoritmo perante um problema como um jogo de *Sudoku* permitiu-nos concluir tanto a eficiência como o próprio procedimento único que cada um destes algoritmos segue para chegar a um objetivo final.

# Referências

- [1] Wolfgang Ertel, Introduction to Artificial Intelligence (Undergraduate Topics in Computer Science), Global Edition 4th ed. Springer, 2017.
- [2] El-Ghazali Talbi, Metaheuristics From Design to Implementation. Pearson, 2009.
- [3] Ivan Bratko, *Prolog Programming for Artificial Intelligence 4th ed.* Addison-Wesley, 2011.